Project 2 – Postfix Translator

In this project, we are implementing a GNU assembly language program that interprets a single line of postfix expression involving decimal quantities and outputs and equivalent RISC-V 32-bit machine language instructions.

Our code generates the RISC-V instructions following the algorithm below:

- **Number Handling**: Push each numeric value onto the stack until an operator is encountered.
- **Operator Encounter**: Upon encountering an operator, pop the top two elements. Assign the first popped element to the x2 register and the second to the x1 register.
- **Operation Execution**: Execute the operation with x1 as the destination register.
- **Immediate Loading**: Utilize the addi rd, x0, imm instruction format for loading values into registers, where x0 is a constant zero register.

Here is a table showing the conversion from RISC-V assembly code to machine instructions:

RISC-V assembly code	RISC-V machine instructions
`add rd, rs1, rs2`	Function (funct7): 0000000 (Addition) rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 (Addition) rd (Destination Register): 5 bits Opcode: 0110011 (same for all R types)
`sub rd, rs1, rs2`	Function (funct7): 0100000 rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0110011
`mul rd, rs1, rs2`	Function (funct7): 0000001 rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0110011
`xor rd, rs1, rs2`	Function (funct7): 0000100 rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0110011
`and rd, rs1, rs2`	Function (funct7): 0000111 rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0110011
`or rd, rs1, rs2`	Function (funct7): 0000110 rs2 (Source Register 2): 5 bits rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0110011
`addi rd, rs1, 5`	Immediate (12 bit binary number): 000000000101 rs1 (Source Register 1): 5 bits Function (funct3): 000 rd (Destination Register): 5 bits Opcode: 0010011 (I format)

To understand the task better we give an example in the beginning:

To the input ***2 3 +*** our program gives the following output:

00000000011 00000 000 00010 0010011
00000000010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011

Let's break down this output to understand it better.

- **00000000011 00000 000 00010 0010011:**

  This line represents the **addi, rd, rs1, imm** instruction, where we have **3** as the **immediate** value, **x2** as the destination register.

- **00000000010 00000 000 00001 0010011:**

  This line represents the **addi, rd, rs1, imm** instruction, where we have **2** as the **immediate** value, **x1** as the destination register.

- **0000000 00010 00001 000 00001 0110011:**

  This line represents the **add, rd, rs1, rs2** instruction. We are getting values from **x2** and **x1** and putting them in the **destination register x1**.

So, this output means the following: In RISC-V assembly language, we would need to do these 3 instructions to calculate the value **2 + 3**. The output is those assembly instructions, converted to machine instructions.

Here is how we achieved it:

We are processing the input characters one by one. Let's call them tokens. And to make it clear from the beginning, only correct postfix expressions are handled. Erroneous expressions are out of the scope of this project.

As described in the beginning, we are checking first checking if the current token is **\n**. If so, we are ending the parsing process and printing the output. Obviously if the input is empty. We immediately exit the process and quit, resulting in empty output. Then we are checking if the token is one of the following: ***, +, -, |, &, ^**. If so, we are invoking the operation performing section. If none of these conditions are satisfied, we say the token must be a number. Here is how this is implemented:

```
parse_loop:
    mov $0, %rbx
    movb (%r14), %bl  # Load next character

    # Check if the char is null terminator
    cmp $'\n', %rbx
    je end_parsing

    # Check if it is whitespace
    cmp $0x20, %bl
    je next_char

    cmp $0x2a, %bl
    je perform_mul

    cmp $0x2b, %bl
    je perform_addition

    cmp $0x2d, %bl
    je perform_sub

    cmp $0x7c, %bl
    je perform_or

    cmp $0x26, %bl
    je perform_and

    cmp $0x5e, %bl
    je perform_xor

    jmp parse_token
```

To describe the further process let's go through with an example: **2 3 +**

When we see the **+** operator we pop the last 2 values from the stack. That gives us 2 and 3 because they were pushed to the stack when they were read, and they are on the top 2 spot. We then to the actual addition and find 5. We push it back to the stack. We also write the necessary output to the output buffer as explained before. Then we go on.

One of the biggest challenges we encountered was printing the binary values of integers. Since the way we append the output buffer was adding bits one by one, we tried to extract each of 12 bits. In order to do that, we initially set the value of a register to 2048, which is the 11^{th} power of 2. For each loop we put this value and the number into the bitwise and operator. If the result is 0, then 0 bit is added to output buffer. If the result is equal to value (00..100..) then 1 bit is added to output buffer. Then we divide the value by 2 (shifting right), to calculate next bit. When the value equals to 0, the loop ends and we added all the 12 bits to output buffer. Here is how it looks:

```

bloop1:
    cmp $0, %rax
    je end_bloop1

    mov %rax, %r9

    and %r12, %r9

    cmp $0, %r9
    je add_zero1

    cmp %rax, %r9
    je add_one1

```

If we read a **7 *** after the previous ones. We again try to pop 2 values of the stack which gives us 5 and 7. Then we do the multiplication process as explained above.

Usage:

When the project is cloned to your machine. You can do the following to try the program to for example see the result for the expression (2 +3) * (4 + 5):

```
$ make
$ ./postfix_translator
2 3 + 4 5 + *
```

You would get the output:

```
000000000011 00000 000 00010 0010011
000000000010 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000000101 00000 000 00010 0010011
000000000100 00000 000 00001 0010011
0000000 00010 00001 000 00001 0110011
000000001001 00000 000 00010 0010011
000000000101 00000 000 00001 0010011
0000001 00010 00001 000 00001 0110011
```